

# A Revision of the Mool Language

Cláudio Vasconcelos      António Ravara

NOVA-LINCS and Dep. de Informática, FCT.  
Universidade NOVA de Lisboa, Portugal

September 23, 2016

## Abstract

We present here in a thorough analysis of the Mool language, covering not only its implementation but also the formalisation (syntax, operational semantics, and type system). The objective is to detect glitches in both the implementation and in the formal definitions, proposing as well new features and added expressiveness. To test our proposals we implemented the revision developed in the Racket platform.

## 1 Introduction

This article presents an analysis of the Mool language, a small object-oriented language similar to Java, developed by Campos and Vasconcelos [1, 2]. The language allows to associate with each class a behavioural type specifying safe orderings of method calls, along the lines of [5].

This analysis is a contribution to the development of the language, detecting bugs not only in the implementation, but also in formalisation. We also propose revisions of aspects of the language we find too restrictive.

Section 2 presents correction proposals. We organise them in two categories: minor aspects (Section 2.1), which have little influence on the language or their correction is very straightforward; and major aspects in (Section 2.2), which heavily influence the behaviour of the language and are more complex to change.

We complement the analysis of the Mool language formal system with a small review of the Mool compiler (version 0.3, available in May 2016 from `gloss.di.fc.ul.pt/mool/download`). The purpose is to understand if the aspects we presented in section 2 were solved in the implementation, and, if they were, how the compiler copes with them.

To test our analysis, we implemented the original formalisation of Mool using PLT-Redex [3], a module available in Racket [4] that allows us to implement and debug formal systems of programming languages. Section 4 present our implementation and explains briefly the examples we used to demonstrate how the aspects in Section 2 affect the language. The code of our implementation, along with the examples, is available at <https://sourceforge.net/p/mool-plt-redex/code/ci/master/tree/mool1.rkt>.

Section 5 consists on our revision proposal for the Mool language. We present a full formal system, consisting on the revised operational semantics and a type

system of the language, based on the original but with changes that try to solve the aspects identified in 2 plus the addition of new features such as constructors.

Again, to test our revision we implemented the revised formalisation using PLT-Redex. Section 6 presents the list of examples used to test this second implementation. Most of these examples are almost identical to the ones in section 4, but now they are expected to have a different behaviour, while a few new examples that were used to test our changes a little further. The code of the implementation, along with the examples, is available at <https://sourceforge.net/p/mool-plt-redex/code/ci/master/tree/mool2.rkt>.

## 2 The original Mool language

Like said before, the main objective is to understand where Mool can be too restrictive or even present incorrect behaviour. We did this by not only reviewing the original definitions [2], but also by implementing the language using PLT Redex and trying to falsify properties of the system (see Section 4). These aspects have been categorised in major and minor aspects, based on their complexity.

### 2.1 Minor errors and limitations

The following observations are minor errors and limitations found on Mool, i.e., they are very simple to solve:

1. The evaluation context for `while` is unnecessary. The evaluation contexts defined in the syntax of Mool specify that in a `while` expression the expression  $e$  that serves as the boolean condition must be evaluated before the `while` expression itself, but the reduction rule R-WHILE specifies that a `while` expression should be immediately reduced to a `if – else` expression.
2. T-USAGEVAR returns a new typing environment but it is not clear why the final environment needs to be different from the initial.
3. T-ASSIGN restricts assignments to unrestricted variables and fields only, but assignment to linear variables can be possible since any case that can risk linearity can be prevented by a predicate that checks if a variable has a linear type when it should not (for example, that already happens in rule T-CLASS where is specified that all of the class fields should be unrestricted).
4. T-CALL specifies that the parameter type should be the same as the method type, which is unnecessarily restricting.

### 2.2 Major errors and limitations

The following aspects are errors and limitations found on Mool that are more complex to solve:

1. Subtyping for variant types is not well defined. The correct definition, based on the sub-typing definition in [5], is as follows:

$$\text{If } \langle u' + u'' \rangle <: u \text{ then } u = \langle u_t + u_f \rangle \text{ with } u' <: u_t \text{ and } u'' <: u_f$$

2. Subtyping seems to be unsafe. Consider the following expression:

$$\text{if}(f.\text{eof}()) \{ f.\text{close}(); \text{false}; \} \text{ else } \{ f.\text{read}(); \text{true}; \}$$

In this expression, if the file has been fully read then it closes and returns *false*, informing the client that there is no more lines to read, otherwise it reads a line and returns *true*, informing the client that there are still lines to be read. Assume that we reverse the result output as follow:

$$\text{if}(f.\text{eof}()) \{ f.\text{close}(); \text{true}; \} \text{ else } \{ f.\text{read}(); \text{false}; \}$$

Mool accepts this, but it can cause a runtime error because the client can try to close an already closed file. In this revision we will not propose a fix for the subtyping since it is not in the context of our work.

3. The typing rule T-SPAWN states that the expression  $e$  should have an unrestricted type, but that is not enough to prevent situations where the occurrence of statements being executed in different threads can result in the correct execution flow of a program being disrespected. For example, assuming a *File* class with the usage

$$\text{lin}\{ \text{open} : \text{lin}\{ \text{read} : \text{lin}\{ \text{close} : \text{un}\{ \} \} \} \}$$

where methods *open*, *read* and *close* are all of type *unit*, the code

$$f.\text{open}(); \text{spawn } f.\text{read}(); f.\text{close}()$$

which opens the file, creates a separate thread for the reading operation and closes the file, is wrong because after creating the new thread with the reading operation it is not possible to predict the next step, so the file can be read or closed. As defined, the type system will accept this because *f.read()* has type *unit*, which is an unrestricted type, and so the typing rule T-SPAWN will accept this expression, as the following partial derivation shows:

$$\begin{array}{c}
\frac{\frac{\dots}{\Gamma \triangleright f.\text{open}() : \text{unit} \triangleleft \Gamma'} \text{T-CALL} \quad \text{T1}}{\Gamma \triangleright f.\text{open}(); \text{spawn } f.\text{read}(); f.\text{close}() : \text{unit} \triangleleft \Gamma'''} \text{T-SEQ} \\
\\
\frac{\frac{\frac{\dots}{\Gamma' \triangleright f.\text{read}() : \text{unit} \triangleleft \Gamma''} \text{T-CALL} \quad \text{T2}}{\Gamma' \triangleright \text{spawn } f.\text{read}() : \text{unit} \triangleleft \Gamma''} \text{T-SPAWN} \quad \text{T1}}{\Gamma' \triangleright \text{spawn } f.\text{read}(); f.\text{close}() : \text{unit} \triangleleft \Gamma''} \text{T-SEQ} \\
\\
\text{T2} \quad \frac{\dots}{\Gamma'' \triangleright f.\text{close}() : \text{unit} \triangleleft \Gamma'''} \text{T-CALL}
\end{array}$$

$$\begin{aligned}
\Gamma &= f : File[\text{lin}\{ \text{open} : \text{lin}\{ \text{read} : \text{lin}\{ \text{close} : \text{un}\{ \} \} \} \} ] \\
\Gamma' &= f : File[\text{lin}\{ \text{read} : \text{lin}\{ \text{close} : \text{un}\{ \} \} \} ] \\
\Gamma'' &= f : File[\text{close} : \text{un}\{ \} ] \\
\Gamma''' &= f : File[\text{un}\{ \} ]
\end{aligned}$$

4. Usages allow incorrect specifications of sequence of methods calls. Consider the following usage type:

$$\text{lin}\{ \text{read} : \mu\text{Read}.\text{un}\{ \text{eof} : \langle \text{close} : \text{un}\{ \} \rangle + \text{read} : \text{Read} \} \} \}$$

This usage describes a behaviour for a *File* class, where method *read* depends on variables initialized by a method *open* that is implemented as a private method and is never called, that allows to read a line from a file before opening it but the typechecker allows it.

5. The type checker does not check if a field is initialised or not, allowing these to be dereferenced even when they are not.
6. The type system does not have typing rules for self calls. Although the typing rules for self calls were deliberately omitted from [2], they are essential since in case of recursion, the type system will not terminate the program evaluation. For instance, the method *run* of the class *Seller* of the example presented in Chapter 2 of [2] is an example of a program that contains a self call that causes the type checker to go into an infinite loop.
7. Private methods are not evaluated since the type system, as defined, only checks methods in the class usage, which the system description considers public, and self calls are not included in the type system.
8. Typing rules for the control flow expressions with method calls as conditions are not applied when the method call is preceded of a negation, like

$$\text{if}(!f.\text{eof}()) \{ f.\text{read}() \} \text{ else } \{ f.\text{close}() \}$$

, treating these calls as regular expressions and so it does not operate the necessary usage changes.

9. The language formalisation does not allow unrestricted classes, i.e., classes without usages.
10. *null* cannot be used as a value, not allowing the programmer to set objects to *null* or check if they are *null*.
11. An usage can go from an unrestricted state into a different state. According to the system description, an usage cannot go from an unrestricted state into a linear state.

$$\text{lin}\{ \text{open} : \mu\text{Read}.\text{un}\{ \text{eof} : \langle \text{close} : \text{un}\{ \} \rangle + \text{read} : \text{Read} \} \} \}$$

This usage, presented in the configuration of the core language, is a slightly modified usage to the *File* class of the example presented in [2]. The type system, as defined, will accept this usage but it clearly represents a situation where the usage goes from unrestricted to linear since when executing the method *open* the usage goes from linear to unrestricted and when executing the method *eof* the usage goes back to being linear.

Although, the same concerns are valid when an usage is composed by several unrestricted states and it transits between unrestricted states. Consider a variation of the *FileReader* class that hosts a file whose reading access can be blocked or unblocked. A possible usage would be:

$$\text{lin}\{ \text{open} : \mu\text{Blocked.un}\{ \text{unblock} : \mu\text{Unblocked.un}\{ \text{block}; \text{Blocked} + \text{read} : \text{Unblocked} \} \} \}$$

Consider also a situation where an instance of this *FileReader* class, in state *Unblocked*, is shared between two clients. Since the usage allows concurrent interaction with the instance, it is possible for one client to execute *read* and the other client to execute *block* at the same time and the *block* operation terminates before the *read* operation. The client that is trying to read will do it while the usage is in state *Blocked*, which is not the expected behaviour.

When in an unrestricted state, not only it must no return to a linear state it also must only go to the same state or to an equivalent state (i.e., a state with the exact same actions), like the following example:

$$\text{lin}\{ \text{open} : \mu\text{Blocked.un}\{ \text{push} : \mu\text{Unblocked.un}\{ \text{push} : \text{Blocked} \} \} \}$$

Although the original definition [2] lacked the ability to declare local variables, it was mentioned that the implementation of Mool at the time had allowed it, so this aspect was omitted from this list.

### 3 Latest Mool implementation

The work developed and presented in the following sections is based on the Mool language presented in [2], but we also reviewed the current Mool implementation available <sup>1</sup> to check if the aspects noted in Sections 2.1 and 2.2 still remain or not and try to understand how the language copes with those aspects. The examples used in this section are based on the *FileAll.mool* example.

To check if the subtyping in the current version is still unsafe, consider the following code:

Listing 1: *FileReader* subtyping example

```

1      if(f.eof()) {
2          f.close();
3          true;
4      } else {
5          s = s ++ f.read();
6          false;
7      }

```

<sup>1</sup>The latest Mool implementation is available at [gloss.di.fc.ul.pt/tryit/Mool](http://gloss.di.fc.ul.pt/tryit/Mool)

While using this code as the body of the method *next* of the *FileReader* class, the compiler accepts it but running it will cause an infinite loop, which not only is a runtime error, it goes against the behaviour specified by the usage since the interaction with the file should be terminated after closing it, but in this example the *FileReader* will execute the methods *eof* and *close*. This proves that subtyping is still unsafe.

The compiler for the current Mool implementation checks if all of class fields are initialised, even if they are not used, instead of waiting for a runtime error, showing that the problem presented in item 6 of Section 2.2 seems to be fixed. The compiler also allows to assign values of linear type to variables, showing that the restriction mentioned in item 3 of Section 2.1 was dropped, allowing code like this:

Listing 2: *FileReader* linear attribution example

```

1      FileReader f; f = new FileReader();
2      f.open();
3
4      FileReader f2; f2 = new FileReader();
5      f2.open();
6
7      f = f2;
```

Moreover, it is possible to observe two aspects of the *spawn* construct: Mool does not allow *e* to be a sequential composition (it must only be a single expression) and not only it must be a method call, it must consume that variable's usage. This last aspect hints that the rule T-SPAWN checks if all variables in the typing environment are unrestricted after executing *e*. Using the example presented in item 2 of Section 2.2, with a class *File* with the following usage:

Listing 3: *File* usage variation

```

1      class File {
2          usage lin{open; Read} where
3              Read = lin{read ; Close}
4              Close = lin{close ; end};
5          ...
6      }
```

The following code, which is identical to the one from the example, will not compile, with the compiler saying that it expected *f* to be *null* in the third line :

Listing 4: *FileReader* spawn example 1

```

1      File f; f = new File();
2      f.open();
3      spawn f.read();
4      f.close();
```

However, the following code will compile, because the method *close* finalises the consumption of *f*'s usage:

Listing 5: *FileReader* spawn example 2

```

1      File f; f = new File();
2      f.open();
3      f.read();
4      spawn f.close();
```

About the unsafe sequence of calls in item 4 of section 2.2, consider the following example:

Listing 6: *File* unsafe usage

```

1      class File {
2          usage lin{read; Read} where
3              Read = lin{eof;
4                  <lin{close; end} + lin{read; Read}>};
5      }

```

Replacing the original usage of the *FileAll.mool* example with the one presented above will result in the program entering an infinite loop, due to the fact that the method *open* is never called, meaning that both variables *linesRead* and *linesInFile* are never explicitly initialized and so both are initialized with the default value which is 0. It is valid to assume that, while the current version of Mool checks if a variable is initialized in the code, it seems to not check if that initialization happens during the execution of the program, leading to these type of situations.

About the use of negated calls as conditions in control flow expressions, the current compiler still has this limitation. The following example will not compile, saying that the method *read* must be called on a control flow expression:

Listing 7: *FileReader* negated call example 1

```

1      if(!f.eof()) {
2          s = s ++ f.read();
3          true;
4      } else {
5          f.close();
6          false;
7      }

```

The message given by the compiler is not very clear since the method *read* is being called inside a control flow expression but the reason for this error is due to the fact that, during the type-checking process, the rule T-IF is applied instead of the rule T-IFV, and it does not operate the necessary changes to the usage of the field *f* so that method *read* is available to be called inside the first branch and the method *close* inside the second. Another example is the following code where a *while* expression is used but the compiler does not accept the code for the same reason as the previous example:

Listing 8: *FileReader* negated call example 2

```

1      while(!f.eof()) {
2          s = f.read() ++ s;
3      }

```

The current compiler allows classes to be unrestricted, as shown by the example *PetitionAll.mool* which has unrestricted classes such as *Main* and *PetitionServer*.

Furthermore, the current compiler does not allow an usage to go from unrestricted to linear. The following example will not compile:

Listing 9: *FileReader* bad usage example 1

```

1      class File {
2          usage lin{open; Read} where

```

```

3           Read = un{eof;
4               <lin{close; end} + lin{read; Read}>};
5       ...
6   }

```

Furthermore, the current compiler does not allow an usage to go from unrestricted to linear. The following example will not compile:

Listing 10: *FileReader* bad usage example 2

```

1   class File {
2       usage lin{open; Read} where
3           Read = un{eof;
4               <lin{close; end} + lin{read; Read}>};
5       ...
6   }

```

But the compiler can accept an usage that goes from an unrestricted state to another different unrestricted state, like the following one:

Listing 11: *FileReader* bad usage example 3

```

1   class FileReader {
2       usage lin{open; Blocked} where
3           Blocked = un{unblock; Unblocked}
4           Unblocked = un{read; Unblocked + block; Blocked};
5       ...
6   }

```

## 4 PLT Redex implementation of the original formalization

We implemented Mool as presented in [2] using PLT Redex<sup>2</sup>. Due to the syntax of Racket, we had to make some modifications on the syntax of Mool, such as:

- Every expression must be in parenthesis.
- `;` is reserved by Racket, so it cannot be used to separate expressions.
- `.` is also reserved by Racket, so it was replaced by `->`.
- To help implementing the type system, the usage variables **X** were replaced by **!X** so they could be distinguished from regular variables.
- A new construct, `getref`, was added to the runtime syntax. This new construct returns the last object identifier created so it can be assigned to a field.
- In the runtime syntax used by the type system, nonterminals  $u$  and  $D$  were added to  $e$  since there must be only one domain which, in this case, is  $e$ .

In addition to the language implementation, the code also contains a few examples to show some of the problems noted in Section 2.2. In order to implement more elaborate examples, some other changes were made:

<sup>2</sup>Available at <https://sourceforge.net/p/mool-plt-redex/code/ci/master/tree/mool1.rkt>



- Items 1, 2 and 4 of Section 2.1 are already solved in the implementation.
- A typing rule for self calls was added. It is the same as T-CALL but it does not change the usage, as the system description specifies.
- Arithmetic and boolean expressions were implemented.

Finally, since `this` does not exist in this version, the object identifier 0 was reserved to represent `this`, so every class field access and self call are done in 0. The examples are the following:

- R-01 Implementation of the File example presented in [2], with an modification on how the program checks if it has reached the end of the file, due to the limitation presented in item 7 of section 2.2. This example serves to test the operational semantics of Mool and when running it the reduction graph of the program's reduction will be shown.
- T-01 Typing example of the File example. When run the type system should be able to check the whole program with success.
- T-02 Typing example that implements the situation expressed in item 2 of Section 2.2. The type checker verifies successfully when it should not.
- T-03 Implementation of the example presented in item 3 of Section 2.2. The type checker evaluates the program successfully even though it is not desirable to have a situation where the file can be closed before being read.
- T-04 Same thing as T-01 but the fields *f* from the *FileReader* class and from the *Main* class are not initialised, while both are dereferenced as in T-01. The program is evaluated successfully, allowing both fields to be dereferenced even though they are not initialised.
- T-05 Same thing as T-01 but in the usage of the *File* class the method *open* is replaced by the method *read*, same as the usage presented in item 4 of Section 2.2. The usage allows to read the file without opening it but the typechecker verifies the program successfully.
- T-06 A variation of the *File* example where the body of the method *count* is changed to *true*. The return type of the method is *unit* but the body of the method is of type *boolean* and the type checker verifies the program successfully since the body of the method is not verified, only its signature;
- T-07 Implementation of the *File* and *FileReader* classes as presented in [2], including the using of a negated method call as a condition for a control flow expression in method *next* of *FileReader*. This example serves to demonstrate the limitation presented in item 8 of Section 2.2 and it should fail.
- T-08 A variation of the *File* and *FileReader* classes, where now the method *next* of *FileReader* reads the whole file at once. This example is to demonstrate again the limitation presented in item 8 of section 2.2 with the same result, but now in a *while* expression.

- T-09 Another typing example that shows that the type system allows an usage to go from unrestricted to linear. This program only contains one class, *File*, but its usage is the same as the first example given in item 11 of Section 2.2.
- T-10 A simplistic version of the *FileReader* where the methods do not do anything but the usage, which is the same as the second usage presented in item 11 of Section 2.2, is composed by two different unrestricted states and they change between them. This should not be allowed but the type checker allows it.

## 5 The revised Mool language

This section presents our revision of the Mool language that tries to solve the problems mentioned in Sections 2.1 and 2.2. Some of the modifications are based on the observations made in Section 3.

### 5.1 Revised syntax

Figure 1 shows a modified syntax for the Mool language. This revised syntax contains the following new/changed elements:

1. Arithmetic and boolean expressions, represented by the nonterminals  $a$  and  $b$  respectively.
2. A new nonterminal  $r$  for value references, which contains local variables  $d$  and  $this$  (to help solving the problems noted in items 5 and 6 in Section 2.2).
3. Expressions  $e$  contain now only values and expressions, including calls, and put the rest of the constructs in a new nonterminal  $s$  that represents statements.
4. Constructs  $g\ d = e$  and  $d = e\ to\ s$  to allow local variable declaration and assignment.
5. Since we want to add the concept of constructor in the language, we modified the construct `new  $C()$`  to `new  $C(e)$` , allowing to pass parameters to the constructor.
6. We divided types into two nonterminals,  $g$  and  $t$ .  $g$  contains types that can be used to declare fields and variables, while  $t$  contains every type in  $g$  plus every other type such as `void` and `null`.
7. We divided the usages into two nonterminals,  $u$  and  $z$ .  $u$  contains the usage constructs that can be used right at the beginning of the usage while  $z$  contains the usage constructs used during compile time. In the runtime syntax we added  $z$  to  $u$  to avoid too many changes to the typing rules.
8. In the nonterminal  $u$  we added  $\epsilon$  to indicate that it is possible to not define an usage, making the class an unrestricted class.

9. The term  $o$ , which are objects identifiers, is moved from the user syntax for the runtime syntax.
10. In the runtime syntax, a new type of value, `null`, is added and it is used to represent values for non initialised objects, and a new type  $C[u; \vec{F}]$ , where  $\vec{F}$  are mappings from fields that are initialised to types, is added to solve the problems in items 4 and 5 of Section 2.2.
11. The evaluation context `while  $\mathcal{E}$   $e$`  is removed for the reasons stated in item 1 of Section 2.1.

## 5.2 Revised operational semantics

Figure 4 shows the modified reduction rules for this revised version of Mool. The rules differ from the original ones, as we add a new environment, *local*, for the local variables.

We modified the rule R-NEW so that it reduces to a sequential composition with the body of the constructor and the created object identifier.

We also add the new rules R-NEWVAR and R-ASSIGNVAR which are for local variable declaration and assignment.

The rule R-ASSIGNFIELDNULL, allows to assign null values to fields, removing them from the object's record.

Figures 5 and 6 show the evaluation functions for the arithmetic and boolean expressions. These functions, based on the ones presented in [6], receive as arguments an expression and both the class field and local variable environment.

## 5.3 Revised type system

In this section we present a new set of typing rules. We omit the unchanged rules with respect to the original system [2].

Figure 8 shows the proposed typing rules for programs:

1. Rule T-CLASS is a modified version of the rule with the same name that has a new premise that checks if the class usage is correct, i.e., it does not go from an unrestricted state to a linear one at any point.

Moreover, evaluation of the usage has an object  $C[u; \emptyset]$  for input, with no declared fields, and a object  $C[u'; \vec{F}]$  for output, forcing the method-level scope of the system. In the end it checks if all fields in  $\vec{F}$  are unrestricted.

2. Rule T-UNCLASS is for unrestricted classes and, instead of verifying the usage, it verifies all of the methods of the class.

We assume that in unrestricted classes every method is independent, i.e., the changes it introduces to the state of the object do not affect other methods (e.g. initialised fields), so every method is verified using the same typing environments.

Figure 9 shows the proposed typing rules for usages:

1. Rule T-BRANCHEND is a variation of T-BRANCH that is applicable when a usage branch terminates and so it does only evaluate the method, not the next usage (because there is none).

### User Syntax

(class declarations)	$D ::= \text{class } C \{u; \vec{F}; \vec{M}\}$
(field declaration)	$F ::= g \ f$
(method declarations)	$M ::= y \ t \ m(t' \ x) \ \{e\}$
(method qualifiers)	$y ::= \epsilon \mid \text{sync}$
(values)	$v ::= \text{unit} \mid n \mid \text{true} \mid \text{false} \mid \text{null}$
(local value references)	$r ::= d \mid \text{this}$
(global value references)	$w ::= r \mid r.f$
(calls)	$c ::= \text{new } C(e) \mid r.m(e) \mid r.f.m(e)$
(arithmetic operations)	$a ::= n \mid w \mid c \mid a + a \mid a - a$ $\mid a * a \mid a / a$
(boolean operations)	$b ::= \text{true} \mid \text{false} \mid w \mid c \mid a == a \mid a != a$ $\mid a <= a \mid a >= a \mid a < a \mid a > a$ $\mid b \ \&\& \ b \mid b \    \ b \mid !b$
(expressions)	$e ::= v \mid a \mid b$ $\mid c \mid w$
(statements)	$s ::= e \mid s; s'$ $\mid r.f = e \mid g \ d = e \mid d = e$ $\mid \text{if } (b) \ s' \ \text{else } s'' \mid \text{while } (b) \{s'\}$ $\mid \text{spawn}\{s\}$
(types)	$t ::= \text{void} \mid g \mid \text{null}$
(declarable types)	$g ::= \text{int} \mid \text{bool} \mid C[z]$
(class usages)	$u ::= \epsilon \mid q\{m_i; z_i\}_{i \in I} \mid \mu X.u$
(usages)	$z ::= u \mid \langle u + u \rangle \mid X$
(usage types)	$q ::= \text{un} \mid \text{lin}$

### Runtime Syntax

(values)	$v ::= \dots \mid o$
(value references)	$r ::= \dots \mid o$
(class usages)	$u ::= \dots \mid z$
(types)	$t ::= \dots \mid C[z; \vec{F}]$
(object records)	$R ::= (C, u, \overrightarrow{f = v}, l)$
(field value map)	$l ::= 0 \mid 1$
(heap)	$h ::= \emptyset \mid h, o = R$
(evaluation context)	$\mathcal{E} ::= [-] \mid \mathcal{E}; s \mid o.f = \mathcal{E} \mid o.m(\mathcal{E}) \mid o.f.m(\mathcal{E})$ $\mid \text{if } (\mathcal{E}) \ s \ \text{else } s'$
(States)	$S ::= (h, \text{local}, s_1 \mid \dots \mid s_n)$

Figure 1: Revised syntax

### Object Record and Heap Operations

$$\begin{aligned}\langle C, u, \vec{V} \rangle . f &\stackrel{\text{def}}{=} \vec{V}(f) & \langle C, u, \vec{V} \rangle . \text{usage} &\stackrel{\text{def}}{=} u \\ \langle C, u, \vec{V} \rangle . \text{class} &\stackrel{\text{def}}{=} C\end{aligned}$$

### Operations for values and types

$$\text{lin}(v) \stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v). \text{usage} = \langle u' + u'' \rangle \\ tt & \text{if } v = o \wedge h(v). \text{usage} = \text{lin}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases} \quad \text{un}(v) \stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = \text{unit} \\ tt & \text{if } v = n \\ tt & \text{if } v = \text{true} \\ tt & \text{if } v = \text{false} \\ tt & \text{if } v = o \wedge h(v). \text{usage} = \text{un}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases}$$

### Class Definition Operations

$$\begin{aligned}C.\text{methods} &\stackrel{\text{def}}{=} \overrightarrow{M, eval} \quad \text{where class } C \{u; \vec{F}; \vec{M}\} \in \vec{D} \text{ and } eval \in \{0, 1\} \\ C.\text{fields} &\stackrel{\text{def}}{=} \vec{F} \quad \text{where class } C \{u; \vec{F}; \vec{M}\} \in \vec{D} \\ C.\text{usage} &\stackrel{\text{def}}{=} u \quad \text{where class } C \{u; \vec{F}; \vec{M}\} \in \vec{D}\end{aligned}$$

Figure 2: Auxiliary definitions and Operations

$$\begin{aligned}\text{R-CONTEXT} &\frac{(h, local, s_1 \mid \dots \mid s \mid \dots \mid s_n) \longrightarrow (h', local', s_1 \mid \dots \mid s' \mid \dots \mid s_n)}{(h, local, s_1 \mid \dots \mid \mathcal{E}[s] \mid \dots \mid s_n) \longrightarrow (h', local', s_1 \mid \dots \mid \mathcal{E}[s'] \mid \dots \mid s_n)} \\ \text{R-SPAWN} &(h, local, s_1 \mid \dots \mid \mathcal{E}[\text{spawn}\{s\}] \mid \dots \mid s_n) \longrightarrow (h, local, s_1 \mid \dots \mid \mathcal{E}[\text{unit}] \mid \dots \mid s_n)\end{aligned}$$

Figure 3: Reduction semantics for states

$$\begin{array}{l}
\text{R-UNFIELD} \quad \frac{h(o).f = v \quad \text{un}(v, h)}{(h, \text{local}, o.f) \longrightarrow (h, \text{local}, v)} \\
\text{R-LINFIELD} \quad \frac{h(o).f = v \quad \text{lin}(v, h)}{(h, \text{local}, o.f) \longrightarrow (h\{o.f \mapsto \text{null}\}, \text{local}, v)} \\
\text{R-UNVAR} \quad \frac{h(d) = v \quad \text{un}(v, h)}{(h, \text{local}, d) \longrightarrow (h, \text{local}, v)} \\
\text{R-LINVAR} \quad \frac{h(d) = v \quad \text{lin}(v, h)}{(h, \text{local}, d) \longrightarrow (h\{d \mapsto \text{null}\}, \text{local}, v)} \\
\text{R-SEQ} \quad (h, \text{local}, v; s) \longrightarrow (h, \text{local}, s) \\
\text{R-NEWVAR} \quad (h, \text{local}, g \ d = v) \longrightarrow (h, \text{local}\{d \mapsto v\}, \text{unit}) \\
\text{R-ASSIGNVAR} \quad (h, \text{local}, d = v) \longrightarrow (h, \text{local}\{d \mapsto v\}, \text{unit}) \\
\text{R-ASSIGNFIELD} \quad \frac{v \neq \text{null}}{(h, \text{local}, o.f = v) \longrightarrow (h\{o.f \mapsto v\}, \text{local}, \text{unit})} \\
\text{R-ASSIGNFIELDNULL} \quad \frac{v = \text{null}}{(h, \text{local}, o.f = v) \longrightarrow (h \setminus o.f, \text{local}, \text{unit})} \\
\text{R-NEW} \quad \frac{o \text{ fresh} \quad (\_ \ C(\_ \ x) \ \{s\}, \_) \in C.\text{methods} \quad C.\text{fields} = \overrightarrow{t \ f} \quad C.\text{usage} = u}{(h, \text{local}, \text{new } C(v)) \longrightarrow ((h, o = \langle C, u, f = \text{null} \rangle), \text{local}, s\{^o/\text{this}\}\{^v/x\}; o)} \\
\text{R-CALL} \quad \frac{(\_ \ m(\_ \ x) \ \{s\}, \_) \in (h(o).\text{class}).\text{methods}}{(h, \text{local}, o.m(v)) \longrightarrow (h, \text{local}, s\{^o/\text{this}\}\{^v/x\})} \\
\text{R-FIELDCALL} \quad \frac{(\_ \ m(\_ \ x) \ \{s\}, \_) \in (h(o).f.\text{class}).\text{methods}}{(h, \emptyset, o.f.m(v)) \longrightarrow (h, \emptyset, s\{^o/\text{this}\}\{^v/x\})} \\
\text{R-WHILE} \quad (h, \text{local}, \text{while } (b)\{s\}) \longrightarrow (h, \text{local}, \text{if } (b) \ (s; \text{while } (b)\{s\}) \ \text{else unit}) \\
\text{R-IFTRUE} \quad (h, \text{local}, \text{if } (\text{true}) \ s' \ \text{else } s'') \longrightarrow (h, \text{local}, s') \\
\text{R-IFFALSE} \quad (h, \text{local}, \text{if } (\text{false}) \ s' \ \text{else } s'') \longrightarrow (h, \text{local}, s'')
\end{array}$$

Figure 4: Revised reduction semantics for statements

$$\begin{array}{l}
\mathcal{N}(n) = n \quad \mathcal{A}(n, h, \text{local}) = \mathcal{N}(n) \\
\mathcal{A}(o.f, h, \text{local}) = h(o).f \quad \mathcal{A}(d, h, \text{local}) = \text{local}(d) \\
\mathcal{A}(a_1 + a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) + \mathcal{A}(a_2, h, \text{local}) \\
\mathcal{A}(a_1 - a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) - \mathcal{A}(a_2, h, \text{local}) \\
\mathcal{A}(a_1 * a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) * \mathcal{A}(a_2, h, \text{local}) \\
\mathcal{A}(a_1 / a_2, h, \text{local}) = \mathcal{A}(a_1, h, \text{local}) / \mathcal{A}(a_2, h, \text{local})
\end{array}$$

Figure 5: Evaluation functions for arithmetic values and expressions

$$\begin{aligned}
\mathcal{B}(\text{true}, h, local) &= \text{true} & \mathcal{B}(\text{false}, h, local) &= \text{false} \\
\mathcal{B}(o.f, h, local) &= h(o).f & \mathcal{B}(d, h, local) &= local(d) \\
\mathcal{B}(a_1 == a_2, h, local) &= \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) = \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) \neq \mathcal{A}(a_2, h, local) \end{cases} \\
\mathcal{B}(a_1 \neq a_2, h, local) &= \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) \neq \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) = \mathcal{A}(a_2, h, local) \end{cases} \\
\mathcal{B}(a_1 < a_2, h, local) &= \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) \geq \mathcal{A}(a_2, h, local) \end{cases} \\
\mathcal{B}(a_1 < a_2, h, local) &= \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) \leq \mathcal{A}(a_2, h, local) \end{cases} \\
\mathcal{B}(a_1 \leq a_2, h, local) &= \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) \leq \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) > \mathcal{A}(a_2, h, local) \end{cases} \\
\mathcal{B}(a_1 \geq a_2, h, local) &= \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) \geq \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \end{cases} \\
\mathcal{B}(b_1 \ \&\& \ b_2, h, local) &= \begin{cases} \text{true} & \mathcal{B}(b_1, h, local) = \text{true} \wedge \mathcal{B}(b_2, h, local) = \text{true} \\ \text{false} & \text{otherwise} \end{cases} \\
\mathcal{B}(b_1 \ || \ b_2, h, local) &= \begin{cases} \text{true} & \mathcal{B}(b_1, h, local) = \text{true} \vee \mathcal{B}(b_2, h, local) = \text{true} \\ \text{false} & \text{otherwise} \end{cases} \\
\mathcal{B}(!b, h, local) &= \begin{cases} \text{true} & \mathcal{B}(b, h, local) = \text{false} \\ \text{false} & \mathcal{B}(b, h, local) = \text{true} \end{cases}
\end{aligned}$$

Figure 6: Evaluation functions for boolean values and expressions

2. Rule T-USAGEVAR returns the same typing environment mapped to the usage variable, reflecting the observation made in the item 2 of 2.1.

Figures 10 and 11 show the typing rules for the arithmetic and boolean expressions. While the syntax itself already enforces the correct types, we need these rules because the operands can change the usage, e.g., a call made on a field as an operand.

Figure 12 shows the proposed typing rules for field and variable dereference, where we added a new rule, T-NULLFIELD, for dereference of fields that have not initialized.

Figure 13 shows the proposed typing rules for simple statements:

1. Two new rules, T-NEWVAR and T-ASSIGNVAR, for local variable declaration and assignment respectively, are added so the type checker can evaluate local variable declarations.

Both T-ASSIGNVAR and T-ASSIGNFIELD allow linear type value assignment, solving the limitation in item 3 of Section 2.1.

2. The rule T-SPAWN is modified based on the conclusions presented in Section 3, making the type checker checking that:
  - (a) all variables modified in  $s$  are unrestricted; and
  - (b) in case of variables that are objects, the usage is fully consumed and therefore cannot be called in any other expression outside the `spawn`;
  - (c) instead of checking if  $s$  has unrestricted type, it allows expressions other than method calls and it also allows  $s$  to be a sequential composition.

Figure 14 shows the proposed typing rules for control flow expressions:

1. Rules T-IFCALL and T-WHILECALL are similar to the original rules T-IFV and T-WHILEV, but we extended them so they can be applied to method calls made on local variables as the conditional expressions for these control flow expressions.

Both these rules are replicated for unrestricted classes through rules T-UNIFCALL and T-UNWHILECALL, with the difference being that there is no usage modification because there is no usage, so they are essentially the rules T-IF and T-WHILE but instead of having a value as a condition they have a call on a object of a unrestricted class.

2. To solve the error in item 7 in section 2.2 we added the rules T-IFNOTCALL and T-WHILENOTCALL, which are similar to the rules T-IFCALL and T-WHILECALL but are for cases where the method call that serves as the condition is negated, resulting in the inverted attribution of the appropriate usage from the variant type given after the verification of the condition to the expressions that compose the control flow expression.
3. Rules T-IF and T-WHILE, which are for cases where the condition is simply a value and not a method call, are similar to the original rules with the same name, but the condition is a value  $v$  instead of an expression  $e$ . All four rules related to the `while` control flow expression were modified so that they allow modifications inside the loop, but to ensure that, in



rules T-WHILECALL and T-WHILENOTCALL, it is possible to execute the condition after executing the loop, both rules state that the type (and, consequently, the usage) of  $w$  after the loop must be the same as the type  $w$  has before executing the condition.

Figure 15 shows the proposed typing rules for method calls:

1. We modified the rule T-NEW so that the constructor is evaluated has a call at the moment of initialisation and added the rule T-UNNEW for unrestricted classes initialisation.
2. Rules T-SELF CALL1 and T-SELF CALL2, which are for method calls made on *this*, are added to solve the problem stated in item 5 in section 2.2.

Unlike the other typing rules for method calls, these do not change the usage, like the system description in [2] specified, and check if the method was already evaluated or not, so that the type checker only checks a method body once in case of self calls, to prevent entering into a loop when the method is recursive. To check this, the *methods* definition presented in 2 is changed so that every method is associated to a boolean operator that informs if the method was already evaluated or not. This operator is ignored in the other method call rules.

3. Rule T-CALL is similar to the original rule with the same name, but it is extended for method calls made on local variables and also with the minor error mentioned in item 4 of 2.1 corrected.

Moreover, due to the definition of the predicate *allows*, in particular the case when the usage is  $\epsilon$ , i.e., the class is unrestricted, the predicate also returns  $\epsilon$ , this rule can also be applied when the call is made on a object of an unrestricted class.

About the subtyping not being safe, one possible solution would be modifying every if-else typing rule to force both branches to be equivalent, i.e., to produce the same changes to the interacted objects. For example, consider the following derivation:

$$\begin{array}{c}
\frac{\Gamma_1 \triangleright f.eof() : \text{bool} \triangleleft \Gamma_2 \quad T1 \quad T2}{\Gamma_1 \triangleright \text{if } (f.eof()) \{ f.close(); \text{true}; \} \text{ else } \{ f.read(); \text{false}; \} : t \triangleleft \Gamma_6} \text{T-IFCALL} \\
\\
\begin{array}{l}
T1 \quad \frac{\frac{\frac{\Gamma_3 \triangleright f.close() : \text{bool} \triangleleft \Gamma_5}{\Gamma_5 \triangleright \text{true} : \text{bool} \triangleleft \Gamma_5} \text{T-TRUE} \quad \text{T-CALL}}{\Gamma_3 \triangleright f.close(); \text{true} : \text{bool} \triangleleft \Gamma_5} \text{T-SEQ} \\
\frac{\Gamma_3 \triangleright f.close(); \text{true} : \text{bool} \triangleleft \Gamma_5}{\Gamma_3 \triangleright f.close(); \text{true} : \text{bool} \triangleleft \Gamma_6} \text{T-INJR}
\end{array} \\
\\
\begin{array}{l}
T2 \quad \frac{\frac{\frac{\Gamma_4 \triangleright f.read() : \text{bool} \triangleleft \Gamma_1}{\Gamma_1 \triangleright \text{true} : \text{bool} \triangleleft \Gamma_1} \text{T-FALSE} \quad \text{T-CALL}}{\Gamma_4 \triangleright f.read(); \text{false} : \text{bool} \triangleleft \Gamma_1} \text{T-SEQ} \\
\frac{\Gamma_4 \triangleright f.read(); \text{false} : \text{bool} \triangleleft \Gamma_1}{\Gamma_4 \triangleright f.read(); \text{false} : \text{bool} \triangleleft \Gamma_6} \text{T-INJL}
\end{array}
\end{array}$$

$$\begin{aligned}
\Gamma_1 &= f : File[Read] \\
\Gamma_2 &= f : File[\langle \text{lin}\{ \text{close} : \text{un}\{ \} \} + \text{lin}\{ \text{read} : Read \} \rangle] \\
\Gamma_3 &= f : File[\text{lin}\{ \text{close} : \text{un}\{ \} \}] \\
\Gamma_4 &= f : File[\text{lin}\{ \text{read} : Read \}] \\
\Gamma_5 &= f : File[\text{un}\{ \}] \\
\Gamma_6 &= f : \langle \Gamma_1 + \Gamma_5 \rangle
\end{aligned}$$

This derivation is of the same example we used to show that subtyping can be unsafe in item 2 of section 2.2, and demonstrates that the type system allows it to be verified. If we remove the subtyping this example would not pass, but neither would any other correct example.

To show how the type system would behave without subtyping For example consider the following usage:

$$\text{lin}\{ \text{open} : \mu Read.\text{un}\{ \text{eof} : \langle \text{close} : \text{un}\{ \} + \text{read} : \text{un}\{ \} \rangle \} \} \}$$

This usage is a variation of the *File* usage, with the difference being that after executing *read* the file is fully read and closes automatically. With this usage, the previous example would work without subtyping:

$$\begin{array}{c}
\frac{\Gamma_1 \triangleright f.\text{eof}() : \text{bool} \triangleleft \Gamma_2 \quad T1 \quad T2}{\Gamma_1 \triangleright \text{if } (f.\text{eof}()) \{ f.\text{close}(); \text{true}; \} \text{ else } \{ f.\text{read}(); \text{false}; \} : t \triangleleft \Gamma_5} \text{T-IFCALL} \\
\\
\begin{array}{c}
\frac{}{\Gamma_3 \triangleright f.\text{close}() : \text{bool} \triangleleft \Gamma_5} \text{T-CALL} \\
\frac{}{\Gamma_5 \triangleright \text{true} : \text{bool} \triangleleft \Gamma_5} \text{T-TRUE} \\
T1 \quad \frac{}{\Gamma_3 \triangleright f.\text{close}(); \text{true} : \text{bool} \triangleleft \Gamma_5} \text{T-SEQ}
\end{array} \\
\\
\begin{array}{c}
\frac{}{\Gamma_4 \triangleright f.\text{read}() : \text{bool} \triangleleft \Gamma_1} \text{T-CALL} \\
\frac{}{\Gamma_1 \triangleright \text{true} : \text{bool} \triangleleft \Gamma_5} \text{T-FALSE} \\
T2 \quad \frac{}{\Gamma_4 \triangleright f.\text{read}(); \text{false} : \text{bool} \triangleleft \Gamma_5} \text{T-SEQ}
\end{array}
\end{array}$$

$$\begin{aligned}
\Gamma_1 &= f : File[Read] \\
\Gamma_2 &= f : File[\langle \text{lin}\{ \text{close} : \text{un}\{ \} \} + \text{lin}\{ \text{read} : \text{un}\{ \} \} \rangle] \\
\Gamma_3 &= f : File[\text{lin}\{ \text{close} : \text{un}\{ \} \}] \\
\Gamma_4 &= f : File[\text{lin}\{ \text{read} : \text{un}\{ \} \}] \\
\Gamma_5 &= f : File[\text{un}\{ \}] \\
\Gamma_6 &= f : \langle \Gamma_1 + \Gamma_5 \rangle
\end{aligned}$$

Although this would work, it can be too restrictive to force both branches to leave the interacted object with the same usage in both returned environments. Because of this and the fact that proposing a more appropriate solution requires a deeper study that is out of the context of our work, we choose to ignore from now on.

### Type Operations

$$\begin{aligned}
\text{lin}(t) &\stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v).\text{usage} = \langle u' + u'' \rangle \\ tt & \text{if } v = o \wedge h(v).\text{usage} = \text{lin}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases} \\
\text{un}(t) &\stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = \text{void} \\ tt & \text{if } v = \text{int} \\ tt & \text{if } v = \text{bool} \\ tt & \text{if } v = o \wedge h(v).\text{usage} = \text{un}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases} \\
\text{lin}(\Gamma) &\stackrel{\text{def}}{=} \forall (t \ f) \in \Gamma : \text{lin}(t) \\
\text{un}(\Gamma) &\stackrel{\text{def}}{=} \forall (t \ f) \in \Gamma : \text{un}(t) \\
\text{lin}(\vec{F}) &\stackrel{\text{def}}{=} \forall (t \ f) \in \vec{F} : \text{lin}(t) \\
\text{un}(\vec{F}) &\stackrel{\text{def}}{=} \forall (t \ f) \in \vec{F} : \text{un}(t) \\
\text{check}(\Phi, u) &\stackrel{\text{def}}{=} \begin{cases} \text{check}(\Phi, u_i) & u = \text{lin}\{m_i; u_i\}_{i \in I} \\ \text{check}(\Phi, u_t) \wedge \text{check}(\Theta, u_f) & u = \langle u_t + u_t \rangle \\ \text{check}((\Phi, X : u), u) & u = \mu X. u \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = \mu X. u_X \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \text{lin}\{m_j; u_j\}_{j \in J} \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \Phi(X) = \langle u_t + u_t \rangle \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \Phi(X) = \text{lin}\{m_j; u_j\}_{j \in J} \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \langle u_t + u_t \rangle \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \text{un}\{m_j; u_j\}_{j \in J} \wedge m_i \neq m_j \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \Phi(X) = \text{un}\{m_j; u_j\}_{j \in J} \wedge m_i \neq m_j \\ tt & \text{otherwise} \end{cases} \\
u.\text{allows}(m_j) &\stackrel{\text{def}}{=} \begin{cases} \epsilon & u = \epsilon \\ \text{un} & u = \text{un} \\ u_j & u = \{m_i; u_i\}_{i \in I} \text{ and } j \in I \\ u' \{ \mu X. u' /_X \}.\text{allows}(m_j) & u = \mu X. u' \\ \text{undefined} & \text{otherwise} \end{cases} \\
\text{agree}(t, t') &\stackrel{\text{def}}{=} \begin{cases} tt & \text{if } t = t' = \text{bool} \\ tt & \text{if } t = t' = \text{int} \\ tt & \text{if } t = t' = \text{void} \\ tt & \text{if } t = C[u] \text{ and } t' = C[u; \vec{F}] \\ tt & \text{if } t = \text{null} \text{ or } t' = \text{null} \end{cases} \\
\text{modified}(\Gamma, \Gamma') &\stackrel{\text{def}}{=} \forall r \in \Gamma' : r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r) \\
\text{completed}(\Gamma, \Gamma') &\stackrel{\text{def}}{=} \forall r \in \Gamma' : (r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r)) \wedge (r = C[u; \vec{F}] \implies u = \text{un}\{\})
\end{aligned}$$

Figure 7: Types, Type Definitions and Operations

$$\begin{array}{c}
\text{T-CLASS} \quad \frac{\text{check}(\emptyset, u) \quad C[u; \emptyset] \triangleright u \triangleleft C[u; \vec{F}] \quad \text{un}(\vec{F})}{\vdash \text{class } C \{u; \vec{F}; \vec{M}\}} \\
\\
\text{T-UNCLASS} \quad \frac{\forall i \in I. \quad (y \ t \ m_i(t' \ x) \ \{s\}, \_) \in \vec{M} \quad C[\emptyset], x : t' \triangleright s \triangleleft C[\vec{F}] \quad \text{un}(\vec{F})}{\vdash \text{class } C\{\vec{F}; \vec{M}\}}
\end{array}$$

Figure 8: Revised typing rules for programs

$$\begin{array}{c}
\text{T-BRANCH} \quad \frac{\forall i \in I. \quad (y \ t \ m_i(t' \ x) \ \{s\}, \_) \in C.\text{methods} \quad \text{this} : C[u; \vec{F}], x : t' \triangleright s : t \triangleleft \Gamma \quad \Gamma \triangleright x : t'' \quad \text{un}(t'') \quad \Gamma \triangleright \text{this} : C[u_i; \vec{F}_i] \quad \Theta; \Gamma \triangleright u_i \triangleleft \Gamma'}{\Theta; C[u; \vec{F}] \triangleright \_ \{m_i; u_i\}_{i \in I} \triangleleft \Gamma'} \\
\\
\text{T-BRANCHEND} \quad \Theta; \Gamma \triangleright \text{un}\{\} \triangleleft \Gamma \quad \text{T-USAGEVAR} \quad (\Theta, X : \Gamma); \Gamma \triangleright X \triangleleft \Gamma
\end{array}$$

Figure 9: Revised typing rules for usages

$$\begin{array}{c}
\text{T-ADD} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 + a_2 : \text{int} \triangleleft \Gamma''} \\
\\
\text{T-SUB} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 - a_2 : \text{int} \triangleleft \Gamma''} \\
\\
\text{T-MULT} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 * a_2 : \text{int} \triangleleft \Gamma''} \\
\\
\text{T-DIV} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 / a_2 : \text{int} \triangleleft \Gamma''}
\end{array}$$

Figure 10: Revised typing rules for arithmetic expressions

$$\begin{array}{c}
\text{T-EQ} \quad \frac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \quad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \quad \text{agree}(t', t)}{\Gamma \triangleright e_1 == e_2 : \text{bool} \triangleleft \Gamma''} \\
\text{T-DIFF} \quad \frac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \quad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \quad \text{agree}(t', t)}{\Gamma \triangleright e_1! = e_2 : \text{bool} \triangleleft \Gamma''} \\
\text{T-GREATER} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 > a_2 : \text{bool} \triangleleft \Gamma''} \\
\text{T-LESS} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 < a_2 : \text{bool} \triangleleft \Gamma''} \\
\text{T-GTEQUAL} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 \geq a_2 : \text{bool} \triangleleft \Gamma''} \\
\text{T-LEEQUAL} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 \leq a_2 : \text{bool} \triangleleft \Gamma''} \\
\text{T-AND} \quad \frac{\Gamma \triangleright b_1 : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright b_2 : \text{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1 \&\& b_2 : \text{bool} \triangleleft \Gamma''} \\
\text{T-OR} \quad \frac{\Gamma \triangleright b_1 : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright b_2 : \text{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1 || b_2 : \text{bool} \triangleleft \Gamma''} \quad \text{T-NOT} \quad \frac{\Gamma \triangleright b : \text{bool} \triangleleft \Gamma'}{\Gamma \triangleright !b : \text{bool} \triangleleft \Gamma'}
\end{array}$$

Figure 11: Revised typing rules for boolean expressions

$$\begin{array}{c}
\text{T-LINVAR} \quad \frac{\text{lin}(g)}{(\Gamma, r : g) \triangleright r : g \triangleleft \Gamma} \quad \text{T-UNVAR} \quad \frac{\text{un}(g)}{(\Gamma, r : t) \triangleright r : g \triangleleft (\Gamma, r : g)} \\
\text{T-LINFIELD} \quad \frac{\Gamma \triangleright \text{this} : C[u; \vec{F}] \quad \vec{F}(f) = g \quad \text{lin}(g)}{\Gamma \triangleright \text{this}.f : t \triangleleft \Gamma\{\text{this} \mapsto C[u; (\vec{F} \setminus f)]\}} \\
\text{T-UNFIELD} \quad \frac{\Gamma \triangleright \text{this} : C[u; \vec{F}] \quad \vec{F}(f) = t \quad \text{un}(t)}{\Gamma \triangleright \text{this}.f : t \triangleleft \Gamma} \\
\text{T-NULFIELD} \quad \frac{\Gamma \triangleright \text{this} : C[u; \vec{F}] \quad (\_ f) \notin \vec{F}}{\Gamma \triangleright \text{this}.f : \text{null} \triangleleft \Gamma}
\end{array}$$

Figure 12: Revised typing rules for field and variable dereference

$$\begin{array}{c}
\text{T-ASSIGNVAR} \quad \frac{e \neq \text{null} \quad \Gamma \triangleright d : g \triangleleft \Gamma' \quad \Gamma \triangleright e : g' \triangleleft \Gamma' \quad \text{agree}(g', g)}{\Gamma \triangleright d = e : \text{void} \triangleleft \Gamma'} \\
\\
\text{T-ASSIGNFIELD} \quad \frac{e \neq \text{null} \quad \Gamma \triangleright e : g \triangleleft \Gamma' \quad \Gamma' \triangleright \text{this} : C[u; \vec{F}] \quad C.\text{fields}(f) = g' \quad (\_ f) \notin \vec{F} \vee \vec{F}(f) = g \quad \text{agree}(g', g)}{\Gamma \triangleright \text{this}.f = e : \text{void} \triangleleft \Gamma' \{ \text{this} \mapsto C[u; (\vec{F} \cup (g \ f))] \}} \\
\\
\text{T-ASSIGNFIELDNULL} \quad \frac{\Gamma' \triangleright \Gamma \triangleright e : \text{null} \triangleleft \Gamma \quad \text{this} : C[u; \vec{F}]}{\Gamma \triangleright \text{this}.f = e : \text{void} \triangleleft \Gamma' \{ \text{this} \mapsto C[u; \vec{F} \setminus (\_ f)] \}} \\
\\
\text{T-NEWVAR} \quad \frac{d \notin \Gamma \quad \Gamma \triangleright e : g' \triangleleft \Gamma' \quad \text{agree}(g', g)}{\Gamma \triangleright g \ d = e : \text{void} \triangleleft \Gamma' \{ d \mapsto g \}} \\
\\
\text{T-SPAWN} \quad \frac{\Gamma \triangleright s : t \triangleleft \Gamma' \quad \text{un}(\text{modified}(\Gamma, \Gamma')) \quad \text{completed}(\Gamma, \Gamma')}{\Gamma \triangleright \text{spawn } \{s\} : \text{void} \triangleleft \Gamma'}
\end{array}$$

Figure 13: Revised typing rules for simple statements

$$\begin{array}{c}
\text{T-IFCALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_t; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma' \{ w \mapsto C[u_f; \vec{F}] \} \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (w.m(e)) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-IFNOTCALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_f; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma' \{ w \mapsto C[u_t; \vec{F}] \} \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (!w.m(e)) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-IFUNCALL} \quad \frac{\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\epsilon; \vec{F}] \quad \Gamma' \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma' \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (w.m(e)) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-IF} \quad \frac{\Gamma \triangleright b : \text{bool} \triangleleft \Gamma \quad \Gamma \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (b) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-WHILECALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_t; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma(w) = \Gamma''(w)}{\Gamma \triangleright \text{while } (w.m(e)) \{s'\} : t \triangleleft \Gamma'' \{ w \mapsto C[u_f; \vec{F}] \}} \\
\\
\text{T-WHILENOTCALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_f; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma(w) = \Gamma''(w)}{\Gamma \triangleright \text{while } (!w.m(e)) \{s'\} : t \triangleleft \Gamma'' \{ w \mapsto C[u_t; \vec{F}] \}} \\
\\
\text{T-WHILEUNCALL} \quad \frac{\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\epsilon; \vec{F}] \quad \Gamma' \triangleright s : t \triangleleft \Gamma}{\Gamma \triangleright \text{while } (w.m(e)) \{s\} : t \triangleleft \Gamma'} \\
\\
\text{T-WHILE} \quad \frac{\Gamma \triangleright b : \text{bool} \triangleleft \Gamma \quad \Gamma \triangleright s : t \triangleleft \Gamma}{\Gamma \triangleright \text{while } (b) \{s\} : t \triangleleft \Gamma}
\end{array}$$

Figure 14: Revised typing rules for control flow expressions

$$\begin{array}{c}
\text{T-NEW} \quad \frac{C.\text{usage} = \text{lin}\{C; u\} \quad \Gamma \triangleright e : t' \triangleleft \Gamma' \quad (t \ C(t' x) \ \{s\}, \_) \in C.\text{methods} \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright \text{new } C(e) : C[u] \triangleleft \Gamma'} \\
\\
\text{T-UNNEW} \quad \frac{C.\text{usage} = \epsilon \quad \Gamma \triangleright e : t' \triangleleft \Gamma' \quad (t \ C(t' x) \ \{s\}, \_) \in C.\text{methods} \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright \text{new } C(e) : C[u] \triangleleft \Gamma'} \\
\\
\text{T-SELF CALL1} \quad \frac{\Gamma \triangleright e : t' \triangleleft \Gamma' \quad \Gamma' \triangleright \text{this} : C[u; \vec{F}] \quad (t \ m(t' x) \ \{s\}, 0) \in C.\text{methods} \quad \Gamma' \triangleright s : t \triangleleft \Gamma'' \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright \text{this}.m(e) : t \triangleleft \Gamma''} \\
\\
\text{T-SELF CALL2} \quad \frac{\Gamma \triangleright e : t' \triangleleft \Gamma' \quad \Gamma' \triangleright \text{this} : C[u; \vec{F}] \quad (t \ m(t' x) \ \{s\}, 1) \in C.\text{methods}}{\Gamma \triangleright \text{this}.m(e) : t \triangleleft \Gamma'} \\
\\
\text{T-CALL} \quad \frac{w \neq \text{this} \quad u.\text{allows}(m) = u' \quad \Gamma \triangleright e : t' \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[u; \vec{F}] \quad (t \ m(t' x) \ \{s\}, \_) \in C.\text{methods} \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright w.m(e) : t \triangleleft \Gamma' \{w \mapsto C[u'; \vec{G}]\}}
\end{array}$$

Figure 15: Revised typing rules for calls

## 6 PLT Redex implementation of the revised Mool formalization

To test our revision we implemented our formal system in PLT Redex.<sup>3</sup> Some of the examples in this version, aside from local variables and the use of `this` as an value reference, are equal to the ones in the PLT Redex implementation of Mool presented in Section 4. The examples presented in this version are the following:

- R-01 Implementation of the *File* example presented in [2] to test the operational semantics of Mool. Running it will result in the reduction graph of the program's reduction being shown.
- R-01 Implementation of the *File* example presented in [5].
- R-03 Example of a small program that uses an unrestricted class. The program contains the class *Folder* which contains three methods independent from each other and a *Main* class where a object of *Folder* is created and interacted with. The *Main* class could also be unrestricted but we defined it as linear to show the interaction of an unrestricted class through a linear one.
- R-04 Implementation of the *Auction* example presented in [2] that serves as a more complex test to the operational semantics of Mool.
- T-01 Typing example of the *File* example presented in [2]. Should evaluate successfully.

<sup>3</sup>Available at <https://sourceforge.net/p/mool-plt-redex/code/ci/master/tree/mool2.rkt>

- T-02 With the changes made to the T-SPAWN rule, the type checker notices that executing the *read* operation will modify the variable *f* but will not consume its usage, which goes against what is pretended, so the evaluation should fail.
- T-03 This program is similar to the one from T-05 but instead of creating a new thread for a reading operation, two separate threads are created for opening, reading and close separate files. This example, which evaluates successfully, shows that it is possible to use the construct *spawn* with several expressions.
- T-04 A similar example to T-01 but now the body of the method *main* of class *Main* is executed using *spawn*. Although in the end the variable *f* is unrestricted, it still can be used after the *spawn* expression, so it should fail because we changed the T-SPAWN so that every usage modified inside a *spawn* expression should be at a *end*, making it impossible to call any method from the object after the *spawn* expression.
- T-05 The field *file* of class *FileReader* is not initialised but it is used, so the type checker will fail to evaluate because it checks if the field has already been initialised before using it.
- T-06 Same as T-01 but the constructor of the class *File*, where a number of variables are initialized, is replaced by the incrementation of the field *linesRead*, just like the method *read*. The typechecker does not accept this program;
- T-07 The type system now goes inside the body of private methods and verifies them, so in this example the verification will fail because the type checker notices that the return type of method *count* is *void* but the type of the body is *boolean*;
- T-08 Since now the type system is aware of which methods were already evaluated, this time the type checker will not enter in a infinite loop because it will only evaluate the the body of the recursive method *read* of the class *File* once, ignoring its body when reaching the self call and thus evaluating the program successfully.
- T-09 Typing example of the File example presented in [5]. Should evaluate successfully.
- T-10 A variation of the File example where in the method *next* of the *FileReader* class, after closing the file the field *file* is set to null. The type checker verifies the program successfully.
- T-11 With the new rule T-CLASS the type checker will detect that the usage goes from unrestricted to linear when executing the method *eof*, so the evaluation should fail.
- T-12 Again, the new rule T-CLASS also prevents an usage changing between different unrestricted states, so the program verification should fail;
- T-13 This example is similar to the one in T-12 but now the usage can change between equivalent unrestricted states.



- T-14 Typing example of the simple program introduced in R-03. Should evaluate successfully.
- T-15 Typing example of the *Auction* example presented in [2]. Should evaluate successfully.

## 7 Conclusions and further work

Following a detailed analysis of the formal definition and of the implementation of the Mool programming language, we provide the formalisation of a new version of the language with corrections of errors and broader approaches to aspects where the language is too restrictive. We also provide the implementation of the formalisation of both the original and the revised versions using the Racket programming language, more specifically its PLT Redex module, both complemented with examples to help understanding the evolution between versions.

The next stage of our work will be about the inference of usages from programs written in a variation of Mool based on our revised formalisation but it will not have usage annotations. Instead, the programs will be equipped with assertions that we will use to infer the usages.

## References

- [1] Joana Campos and Vasco T. Vasconcelos. “Channels as Objects in Concurrent Object-Oriented Programming”. In: *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-centric Software*. Vol. 69. EPTCS. 2010, pp. 12–28.
- [2] Joana Correia Campos. “Linear and shared objects in concurrent programming”. MA thesis. University of Lisbon, 2010.
- [3] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.
- [4] Matthias Felleisen et al. “The Racket Manifesto.” In: *SNAPL*. Ed. by Thomas Ball et al. Vol. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 113–128. ISBN: 978-3-939897-80-4. URL: <http://dblp.uni-trier.de/db/conf/snapl/snapl2015.html#FelleisenFFKBMT15>.
- [5] Simon J. Gay et al. “Modular Session Types for Objects”. In: *Logical Methods in Computer Science* 11.4 (2015). DOI: 10.2168/LMCS-11(4:12)2015. URL: [http://dx.doi.org/10.2168/LMCS-11\(4:12\)2015](http://dx.doi.org/10.2168/LMCS-11(4:12)2015).
- [6] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 1846286913.

# A Mool syntax

## User Syntax

(class declarations)	$D ::= \text{class } C \{u; \vec{F}; \vec{M}\}$
(field declaration)	$F ::= g \ f$
(method declarations)	$M ::= y \ t \ m(t' \ x) \ \{e\}$
(method qualifiers)	$y ::= \epsilon \mid \text{sync}$
(values)	$v ::= \text{unit} \mid n \mid \text{true} \mid \text{false} \mid \text{null}$
(local value references)	$r ::= d \mid \text{this}$
(global value references)	$w ::= r \mid r.f$
(calls)	$c ::= \text{new } C(e) \mid r.m(e) \mid r.f.m(e)$
(arithmetic operations)	$a ::= n \mid w \mid c \mid a + a \mid a - a$ $\mid a * a \mid a / a$
(boolean operations)	$b ::= \text{true} \mid \text{false} \mid w \mid c \mid a == a \mid a != a$ $\mid a <= a \mid a >= a \mid a < a \mid a > a$ $\mid b \ \&\& \ b \mid b \    \ b \mid !b$
(expressions)	$e ::= v \mid a \mid b$ $\mid c \mid w$
(statements)	$s ::= e \mid s; s'$ $\mid r.f = e \mid g \ d = e \mid d = e$ $\mid \text{if } (b) \ s' \ \text{else } s'' \mid \text{while } (b) \{s'\}$ $\mid \text{spawn}\{s\}$
(types)	$t ::= \text{void} \mid g \mid \text{null}$
(declarable types)	$g ::= \text{int} \mid \text{bool} \mid C[z]$
(class usages)	$u ::= \epsilon \mid q\{m_i; z_i\}_{i \in I} \mid \mu X.u$
(usages)	$z ::= u \mid \langle u + u \rangle \mid X$
(usage types)	$q ::= \text{un} \mid \text{lin}$

## Runtime Syntax

(values)	$v ::= \dots \mid o$
(value references)	$r ::= \dots \mid o$
(class usages)	$u ::= \dots \mid z$
(types)	$t ::= \dots \mid C[z; \vec{F}]$
(object records)	$R ::= (C, u, \overrightarrow{f = v}, l)$
(field value map)	$l ::= 0 \mid 1$
(heap)	$h ::= \emptyset \mid h, o = R$
(evaluation context)	$\mathcal{E} ::= [-] \mid \mathcal{E}; s \mid o.f = \mathcal{E} \mid o.m(\mathcal{E}) \mid o.f.m(\mathcal{E})$ $\mid \text{if } (\mathcal{E}) \ s \ \text{else } s'$
(States)	$S ::= (h, \text{local}, s_1 \mid \dots \mid s_n)$

## B Mool revised operational semantics

### B.1 Auxiliary definitions and Operations

**Object Record and Heap Operations**

$$\begin{aligned}\langle C, u, \vec{V} \rangle.f &\stackrel{\text{def}}{=} \vec{V}(f) & \langle C, u, \vec{V} \rangle.\text{usage} &\stackrel{\text{def}}{=} u \\ \langle C, u, \vec{V} \rangle.\text{class} &\stackrel{\text{def}}{=} C\end{aligned}$$

**Operations for values and types**

$$\text{lin}(v) \stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v).\text{usage} = \langle u' + u'' \rangle \\ tt & \text{if } v = o \wedge h(v).\text{usage} = \text{lin}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases} \quad \text{un}(v) \stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = \text{unit} \\ tt & \text{if } v = n \\ tt & \text{if } v = \text{true} \\ tt & \text{if } v = \text{false} \\ tt & \text{if } v = o \wedge h(v).\text{usage} = \text{un}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases}$$

**Class Definition Operations**

$$\begin{aligned}C.\text{methods} &\stackrel{\text{def}}{=} \overrightarrow{M, eval} \quad \text{where class } C \{u; \vec{F}; \vec{M}\} \in \vec{D} \text{ and } eval \in \{0, 1\} \\ C.\text{fields} &\stackrel{\text{def}}{=} \vec{F} \quad \text{where class } C \{u; \vec{F}; \vec{M}\} \in \vec{D} \\ C.\text{usage} &\stackrel{\text{def}}{=} u \quad \text{where class } C \{u; \vec{F}; \vec{M}\} \in \vec{D}\end{aligned}$$

### B.2 Reduction semantics for states

$$\text{R-CONTEXT} \frac{(h, local, s_1 \mid \dots \mid s \mid \dots \mid s_n) \longrightarrow (h', local', s_1 \mid \dots \mid s' \mid \dots \mid s_n)}{(h, local, s_1 \mid \dots \mid \mathcal{E}[s] \mid \dots \mid s_n) \longrightarrow (h', local', s_1 \mid \dots \mid \mathcal{E}[s'] \mid \dots \mid s_n)}$$

$$\text{R-SPAWN} (h, local, s_1 \mid \dots \mid \mathcal{E}[\text{spawn}\{s\}] \mid \dots \mid s_n) \longrightarrow (h, local, s_1 \mid \dots \mid \mathcal{E}[\text{unit}] \mid \dots \mid s_n)$$

### B.3 Evaluation functions for arithmetic values and expressions

$$\begin{aligned}\mathcal{N}(n) &= n & \mathcal{A}(n, h, local) &= \mathcal{N}(n) \\ \mathcal{A}(o.f, h, local) &= h(o).f & \mathcal{A}(d, h, local) &= local(d) \\ \mathcal{A}(a_1 + a_2, h, local) &= \mathcal{A}(a_1, h, local) + \mathcal{A}(a_2, h, local) \\ \mathcal{A}(a_1 - a_2, h, local) &= \mathcal{A}(a_1, h, local) - \mathcal{A}(a_2, h, local) \\ \mathcal{A}(a_1 * a_2, h, local) &= \mathcal{A}(a_1, h, local) * \mathcal{A}(a_2, h, local) \\ \mathcal{A}(a_1 / a_2, h, local) &= \mathcal{A}(a_1, h, local) / \mathcal{A}(a_2, h, local)\end{aligned}$$

## B.4 Evaluation functions for boolean values and expressions

$$\mathcal{B}(\text{true}, h, local) = \text{true} \quad \mathcal{B}(\text{false}, h, local) = \text{false}$$

$$\mathcal{B}(o.f, h, local) = h(o).f \quad \mathcal{B}(d, h, local) = local(d)$$

$$\mathcal{B}(a_1 == a_2, h, local) = \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) = \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) \neq \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 \neq a_2, h, local) = \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) \neq \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) = \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 < a_2, h, local) = \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) \geq \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 < a_2, h, local) = \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) \leq \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 \leq a_2, h, local) = \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) \leq \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) > \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(a_1 \geq a_2, h, local) = \begin{cases} \text{true} & \mathcal{A}(a_1, h, local) \geq \mathcal{A}(a_2, h, local) \\ \text{false} & \mathcal{A}(a_1, h, local) < \mathcal{A}(a_2, h, local) \end{cases}$$

$$\mathcal{B}(b_1 \ \&\& \ b_2, h, local) = \begin{cases} \text{true} & \mathcal{B}(b_1, h, local) = \text{true} \wedge \mathcal{B}(b_2, h, local) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathcal{B}(b_1 \ || \ b_2, h, local) = \begin{cases} \text{true} & \mathcal{B}(b_1, h, local) = \text{true} \vee \mathcal{B}(b_2, h, local) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathcal{B}(!b, h, local) = \begin{cases} \text{true} & \mathcal{B}(b, h, local) = \text{false} \\ \text{false} & \mathcal{B}(b, h, local) = \text{true} \end{cases}$$

## B.5 Revised reduction semantics for statements

$$\begin{array}{l}
\text{R-UNFIELD} \quad \frac{h(o).f = v \quad \text{un}(v, h)}{(h, \text{local}, o.f) \longrightarrow (h, \text{local}, v)} \\
\\
\text{R-LINFIELD} \quad \frac{h(o).f = v \quad \text{lin}(v, h)}{(h, \text{local}, o.f) \longrightarrow (h\{o.f \mapsto \text{null}\}, \text{local}, v)} \\
\\
\text{R-UNVAR} \quad \frac{h(d) = v \quad \text{un}(v, h)}{(h, \text{local}, d) \longrightarrow (h, \text{local}, v)} \\
\\
\text{R-LINVAR} \quad \frac{h(d) = v \quad \text{lin}(v, h)}{(h, \text{local}, d) \longrightarrow (h\{d \mapsto \text{null}\}, \text{local}, v)} \\
\\
\text{R-SEQ} \quad (h, \text{local}, v; s) \longrightarrow (h, \text{local}, s) \\
\\
\text{R-NEWVAR} \quad (h, \text{local}, g \ d = v) \longrightarrow (h, \text{local}\{d \mapsto v\}, \text{unit}) \\
\\
\text{R-ASSIGNVAR} \quad (h, \text{local}, d = v) \longrightarrow (h, \text{local}\{d \mapsto v\}, \text{unit}) \\
\\
\text{R-ASSIGNFIELD} \quad \frac{v \neq \text{null}}{(h, \text{local}, o.f = v) \longrightarrow (h\{o.f \mapsto v\}, \text{local}, \text{unit})} \\
\\
\text{R-ASSIGNFIELDNULL} \quad \frac{v = \text{null}}{(h, \text{local}, o.f = v) \longrightarrow (h \setminus o.f, \text{local}, \text{unit})} \\
\\
\text{R-NEW} \quad \frac{o \text{ fresh} \quad (\_ \ C(\_ \ x) \ \{s\}, \_) \in C.\text{methods} \quad C.\text{fields} = t \ \vec{f} \quad C.\text{usage} = u}{(h, \text{local}, \text{new } C(v)) \longrightarrow ((h, o = \langle C, u, f = \text{null} \rangle), \text{local}, s\{^o/\text{this}\}\{^v/x\}; o)} \\
\\
\text{R-CALL} \quad \frac{(\_ \ m(\_ \ x) \ \{s\}, \_) \in (h(o).\text{class}).\text{methods}}{(h, \text{local}, o.m(v)) \longrightarrow (h, \text{local}, s\{^o/\text{this}\}\{^v/x\})} \\
\\
\text{R-FIELDCALL} \quad \frac{(\_ \ m(\_ \ x) \ \{s\}, \_) \in (h(o).f.\text{class}).\text{methods}}{(h, \emptyset, o.f.m(v)) \longrightarrow (h, \emptyset, s\{^o/\text{this}\}\{^v/x\})} \\
\\
\text{R-WHILE} \quad (h, \text{local}, \text{while } (b)\{s\}) \longrightarrow (h, \text{local}, \text{if } (b) \ (s; \text{while } (b)\{s\}) \ \text{else unit}) \\
\\
\text{R-IFTRUE} \quad (h, \text{local}, \text{if } (\text{true}) \ s' \ \text{else } s'') \longrightarrow (h, \text{local}, s') \\
\\
\text{R-IFFALSE} \quad (h, \text{local}, \text{if } (\text{false}) \ s' \ \text{else } s'') \longrightarrow (h, \text{local}, s'')
\end{array}$$

## C Mool revised type system

### C.1 Type operations

#### Type Operations

$$\begin{aligned}
\text{lin}(t) &\stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = o \wedge h(v).\text{usage} = \langle u' + u'' \rangle \\ tt & \text{if } v = o \wedge h(v).\text{usage} = \text{lin}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases} \\
\text{un}(t) &\stackrel{\text{def}}{=} \begin{cases} tt & \text{if } v = \text{void} \\ tt & \text{if } v = \text{int} \\ tt & \text{if } v = \text{bool} \\ tt & \text{if } v = o \wedge h(v).\text{usage} = \text{un}\{m_i; z_i\}_{i \in I} \\ ff & \text{otherwise} \end{cases} \\
\text{lin}(\Gamma) &\stackrel{\text{def}}{=} \forall (t \ f) \in \Gamma : \text{lin}(t) \\
\text{un}(\Gamma) &\stackrel{\text{def}}{=} \forall (t \ f) \in \Gamma : \text{un}(t) \\
\text{lin}(\vec{F}) &\stackrel{\text{def}}{=} \forall (t \ f) \in \vec{F} : \text{lin}(t) \\
\text{un}(\vec{F}) &\stackrel{\text{def}}{=} \forall (t \ f) \in \vec{F} : \text{un}(t) \\
\text{check}(\Phi, u) &\stackrel{\text{def}}{=} \begin{cases} \text{check}(\Phi, u_i) & u = \text{lin}\{m_i; u_i\}_{i \in I} \\ \text{check}(\Phi, u_t) \wedge \text{check}(\Theta, u_f) & u = \langle u_t + u_t \rangle \\ \text{check}((\Phi, X : u), u) & u = \mu X.u \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = \mu X.u_X \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \text{lin}\{m_j; u_j\}_{j \in J} \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \Phi(X) = \langle u_t + u_t \rangle \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \Phi(X) = \text{lin}\{m_j; u_j\}_{j \in J} \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \langle u_t + u_t \rangle \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u = \text{un}\{m_j; u_j\}_{j \in J} \wedge m_i \neq m_j \\ ff & u = \text{un}\{m_i; u_i\}_{i \in I} \wedge \exists u_n \in u_i : u_n = X \wedge \\ & \Phi(X) = \text{un}\{m_j; u_j\}_{j \in J} \wedge m_i \neq m_j \\ tt & \text{otherwise} \end{cases} \\
u.\text{allows}(m_j) &\stackrel{\text{def}}{=} \begin{cases} \epsilon & u = \epsilon \\ \text{un} & u = \text{un} \\ u_j & u = \{m_i; u_i\}_{i \in I} \text{ and } j \in I \\ u' \{ \mu X.u' / X \}.\text{allows}(m_j) & u = \mu X.u' \\ \text{undefined} & \text{otherwise} \end{cases} \\
\text{agree}(t, t') &\stackrel{\text{def}}{=} \begin{cases} tt & \text{if } t = t' = \text{bool} \\ tt & \text{if } t = t' = \text{int} \\ tt & \text{if } t = t' = \text{void} \\ tt & \text{if } t = C[u] \text{ and } t' = C[u; \vec{F}] \\ tt & \text{if } t = \text{null} \text{ or } t' = \text{null} \end{cases} \\
\text{modified}(\Gamma, \Gamma') &\stackrel{\text{def}}{=} \forall r \in \Gamma' : r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r) \\
\text{completed}(\Gamma, \Gamma') &\stackrel{\text{def}}{=} \forall r \in \Gamma' : (r \notin \Gamma \vee \Gamma(r) \neq \Gamma'(r)) \wedge (r = C[u; \vec{F}] \implies u = \text{un}\{\})
\end{aligned}$$

## C.2 Revised typing rules for programs

$$\text{T-CLASS} \frac{\text{check}(\emptyset, u) \quad C[u; \emptyset] \triangleright u \triangleleft C[u; \vec{F}] \quad \text{un}(\vec{F})}{\vdash \text{class } C \{u; \vec{F}; \vec{M}\}}$$

$$\text{T-UNCLASS} \frac{\forall i \in I. \quad (y \ t \ m_i(t' \ x) \ \{s\}, \_) \in \vec{M} \quad C[\emptyset], x : t' \triangleright s \triangleleft C[\vec{F}] \quad \text{un}(\vec{F})}{\vdash \text{class } C\{\vec{F}; \vec{M}\}}$$

## C.3 Revised typing rules for usages

$$\text{T-BRANCH} \frac{\forall i \in I. \quad (y \ t \ m_i(t' \ x) \ \{s\}, \_) \in C.\text{methods} \quad \text{this} : C[u; \vec{F}], x : t' \triangleright s : t \triangleleft \Gamma \quad \Gamma \triangleright x : t'' \quad \text{un}(t'') \quad \Gamma \triangleright \text{this} : C[u_i; \vec{F}_i] \quad \Theta; \Gamma \triangleright u_i \triangleleft \Gamma'}{\Theta; C[u; \vec{F}] \triangleright \_ \{m_i; u_i\}_{i \in I} \triangleleft \Gamma'}$$

$$\begin{array}{ll} \text{T-BRANCHEND} & \Theta; \Gamma \triangleright \text{un}\{\} \triangleleft \Gamma \\ \text{T-USAGEVAR} & (\Theta, X : \Gamma); \Gamma \triangleright X \triangleleft \Gamma \\ \text{T-VARIANT} & \frac{\Theta; \Gamma' \triangleright u_t \triangleleft \Gamma \quad \Theta; \Gamma'' \triangleright u_f \triangleleft \Gamma}{\Theta; \langle \Gamma' + \Gamma'' \rangle \triangleright \langle u_t + u_f \rangle \triangleleft \Gamma} \quad \text{T-REC} \quad \frac{(\Theta, X : \Gamma); \Gamma \triangleright u \triangleleft \Gamma'}{\Theta; \Gamma \triangleright \mu X. u \triangleleft \Gamma'} \end{array}$$

## C.4 Revised typing rules for values

$$\begin{array}{ll} \text{T-UNIT} & \Gamma \triangleright \text{unit} : \text{void} \triangleleft \Gamma \\ \text{T-INT} & \Gamma \triangleright n : \text{int} \triangleleft \Gamma \\ \text{T-TRUE} & \Gamma \triangleright \text{true} : \text{boolean} \triangleleft \Gamma \\ \text{T-FALSE} & \Gamma \triangleright \text{false} : \text{boolean} \triangleleft \Gamma \\ \text{T-NULL} & \Gamma \triangleright \text{null} : \text{null} \triangleleft \Gamma \end{array}$$

## C.5 Revised typing rules for arithmetic expressions

$$\text{T-ADD} \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 + a_2 : \text{int} \triangleleft \Gamma''}$$

$$\text{T-SUB} \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 - a_2 : \text{int} \triangleleft \Gamma''}$$

$$\text{T-MULT} \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 * a_2 : \text{int} \triangleleft \Gamma''}$$

$$\text{T-DIV} \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 / a_2 : \text{int} \triangleleft \Gamma''}$$

## C.6 Revised typing rules for boolean expressions

$$\begin{array}{c}
\text{T-EQ} \quad \frac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \quad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \quad \text{agree}(t', t)}{\Gamma \triangleright e_1 == e_2 : \text{bool} \triangleleft \Gamma''} \\
\\
\text{T-DIFF} \quad \frac{\Gamma \triangleright e_1 : t \triangleleft \Gamma' \quad \Gamma' \triangleright e_2 : t' \triangleleft \Gamma'' \quad \text{agree}(t', t)}{\Gamma \triangleright e_1 ! = e_2 : \text{bool} \triangleleft \Gamma''} \\
\\
\text{T-GREATER} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 > a_2 : \text{bool} \triangleleft \Gamma''} \\
\\
\text{T-LESS} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 < a_2 : \text{bool} \triangleleft \Gamma''} \\
\\
\text{T-GTEQUAL} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 \geq a_2 : \text{bool} \triangleleft \Gamma''} \\
\\
\text{T-LEEQUAL} \quad \frac{\Gamma \triangleright a_1 : \text{int} \triangleleft \Gamma' \quad \Gamma' \triangleright a_2 : \text{int} \triangleleft \Gamma''}{\Gamma \triangleright a_1 \leq a_2 : \text{bool} \triangleleft \Gamma''} \\
\\
\text{T-AND} \quad \frac{\Gamma \triangleright b_1 : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright b_2 : \text{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1 \&\& b_2 : \text{bool} \triangleleft \Gamma''} \\
\\
\text{T-OR} \quad \frac{\Gamma \triangleright b_1 : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright b_2 : \text{bool} \triangleleft \Gamma''}{\Gamma \triangleright b_1 || b_2 : \text{bool} \triangleleft \Gamma''} \quad \text{T-NOT} \quad \frac{\Gamma \triangleright b : \text{bool} \triangleleft \Gamma'}{\Gamma \triangleright !b : \text{bool} \triangleleft \Gamma''}
\end{array}$$

## C.7 Revised typing rules for field and variable dereference

$$\begin{array}{c}
\text{T-LINVAR} \quad \frac{\text{lin}(g)}{(\Gamma, r : g) \triangleright r : g \triangleleft \Gamma} \quad \text{T-UNVAR} \quad \frac{\text{un}(g)}{(\Gamma, r : t) \triangleright r : g \triangleleft (\Gamma, r : g)} \\
\\
\text{T-LINFIELD} \quad \frac{\Gamma \triangleright \text{this} : C[u; \vec{F}] \quad \vec{F}(f) = g \quad \text{lin}(g)}{\Gamma \triangleright \text{this}.f : t \triangleleft \Gamma\{\text{this} \mapsto C[u; (\vec{F} \setminus f)]\}} \\
\\
\text{T-UNFIELD} \quad \frac{\Gamma \triangleright \text{this} : C[u; \vec{F}] \quad \vec{F}(f) = t \quad \text{un}(t)}{\Gamma \triangleright \text{this}.f : t \triangleleft \Gamma} \\
\\
\text{T-NULFIELD} \quad \frac{\Gamma \triangleright \text{this} : C[u; \vec{F}] \quad (\_ f) \notin \vec{F}}{\Gamma \triangleright \text{this}.f : \text{null} \triangleleft \Gamma}
\end{array}$$



## C.8 Revised typing rules for simple statements

$$\begin{array}{c}
\text{T-SEQ} \quad \frac{\Gamma \triangleright s : t \triangleleft \Gamma' \quad \Gamma' \triangleright t' : g' \triangleleft \Gamma''}{\Gamma \triangleright s; s' : t' \triangleleft \Gamma''} \\
\\
\text{T-ASSIGNVAR} \quad \frac{e \neq \text{null} \quad \Gamma \triangleright d : g \triangleleft \Gamma' \quad \Gamma \triangleright e : g' \triangleleft \Gamma' \quad \text{agree}(g', g)}{\Gamma \triangleright d = e : \text{void} \triangleleft \Gamma'} \\
\\
\text{T-ASSIGNFIELD} \quad \frac{e \neq \text{null} \quad \Gamma \triangleright e : g \triangleleft \Gamma' \quad \Gamma' \triangleright \text{this} : C[u; \vec{F}] \quad C.\text{fields}(f) = g' \quad (\_ f) \notin \vec{F} \vee \vec{F}(f) = g \quad \text{agree}(g', g)}{\Gamma \triangleright \text{this}.f = e : \text{void} \triangleleft \Gamma' \{ \text{this} \mapsto C[u; (\vec{F} \cup (g \ f))] \}} \\
\\
\text{T-ASSIGNFIELDNULL} \quad \frac{\Gamma' \triangleright \Gamma \triangleright e : \text{null} \triangleleft \Gamma \quad \text{this} : C[u; \vec{F}]}{\Gamma \triangleright \text{this}.f = e : \text{void} \triangleleft \Gamma' \{ \text{this} \mapsto C[u; (\vec{F} \setminus (\_ f))] \}} \\
\\
\text{T-NEWVAR} \quad \frac{d \notin \Gamma \quad \Gamma \triangleright e : g' \triangleleft \Gamma' \quad \text{agree}(g', g)}{\Gamma \triangleright g \ d = e : \text{void} \triangleleft \Gamma' \{ d \mapsto g \}} \\
\\
\text{T-SPAWN} \quad \frac{\Gamma \triangleright s : t \triangleleft \Gamma' \quad \text{un}(\text{modified}(\Gamma, \Gamma')) \quad \text{completed}(\Gamma, \Gamma')}{\Gamma \triangleright \text{spawn } \{s\} : \text{void} \triangleleft \Gamma'}
\end{array}$$

## C.9 Revised typing rules for control flow expressions

$$\begin{array}{c}
\text{T-IFCALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_t; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma' \{ w \mapsto C[u_f; \vec{F}] \} \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (w.m(e)) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-IFNOTCALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_f; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma' \{ w \mapsto C[u_t; \vec{F}] \} \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (!w.m(e)) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-IFUNCALL} \quad \frac{\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\epsilon; \vec{F}] \quad \Gamma' \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma' \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (w.m(e)) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-IF} \quad \frac{\Gamma \triangleright b : \text{bool} \triangleleft \Gamma \quad \Gamma \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma \triangleright s'' : t \triangleleft \Gamma''}{\Gamma \triangleright \text{if } (b) \ s' \text{ else } s'' : t \triangleleft \Gamma''} \\
\\
\text{T-WHILECALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_t; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma(w) = \Gamma''(w)}{\Gamma \triangleright \text{while } (w.m(e)) \{s'\} : t \triangleleft \Gamma'' \{ w \mapsto C[u_f; \vec{F}] \}} \\
\\
\text{T-WHILENOTCALL} \quad \frac{w \neq \text{this} \quad \Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\langle u_t + u_f \rangle; \vec{F}] \quad \Gamma' \{ w \mapsto C[u_f; \vec{F}] \} \triangleright s' : t \triangleleft \Gamma'' \quad \Gamma(w) = \Gamma''(w)}{\Gamma \triangleright \text{while } (!w.m(e)) \{s'\} : t \triangleleft \Gamma'' \{ w \mapsto C[u_t; \vec{F}] \}} \\
\\
\text{T-WHILEUNCALL} \quad \frac{\Gamma \triangleright w.m(e) : \text{bool} \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[\epsilon; \vec{F}] \quad \Gamma' \triangleright s : t \triangleleft \Gamma}{\Gamma \triangleright \text{while } (w.m(e)) \{s\} : t \triangleleft \Gamma'} \\
\\
\text{T-WHILE} \quad \frac{\Gamma \triangleright b : \text{bool} \triangleleft \Gamma \quad \Gamma \triangleright s : t \triangleleft \Gamma}{\Gamma \triangleright \text{while } (b) \{s\} : t \triangleleft \Gamma}
\end{array}$$

### C.10 Typing rules for subtyping

$$\begin{array}{c}
\text{T-INJL} \frac{\Gamma \triangleright e : t \triangleleft \Gamma'}{\Gamma \triangleright e : t \triangleleft \langle \Gamma' + \Gamma'' \rangle} \quad \text{T-INJR} \frac{\Gamma \triangleright e : t \triangleleft \Gamma''}{\Gamma \triangleright e : t \triangleleft \langle \Gamma' + \Gamma'' \rangle} \\
\\
\text{T-SUB} \frac{\Gamma \triangleright e : C[u] \triangleleft \Gamma' \quad C[u] <: C[u']}{\Gamma \triangleright e : C[u'] \triangleleft \Gamma'} \quad \text{T-SUBENV} \frac{\Gamma \triangleright e : t \triangleleft \Gamma' \quad \Gamma' <: \Gamma''}{\Gamma \triangleright e : t \triangleleft \Gamma''}
\end{array}$$

### C.11 Revised typing rules for calls

$$\begin{array}{c}
\text{T-NEW} \frac{C.\text{usage} = \text{lin}\{C; u\} \quad \Gamma \triangleright e : t' \triangleleft \Gamma' \quad (t \ C(t' x) \ \{s\}, \_) \in C.\text{methods} \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright \text{new } C(e) : C[u] \triangleleft \Gamma'} \\
\\
\text{T-UNNEW} \frac{C.\text{usage} = \epsilon \quad \Gamma \triangleright e : t' \triangleleft \Gamma' \quad (t \ C(t' x) \ \{s\}, \_) \in C.\text{methods} \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright \text{new } C(e) : C[u] \triangleleft \Gamma'} \\
\\
\text{T-SELF CALL1} \frac{\Gamma \triangleright e : t' \triangleleft \Gamma' \quad \Gamma' \triangleright \text{this} : C[u; \vec{F}] \quad (t \ m(t' x) \ \{s\}, 0) \in C.\text{methods} \quad \Gamma' \triangleright s : t \triangleleft \Gamma'' \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright \text{this}.m(e) : t \triangleleft \Gamma''} \\
\\
\text{T-SELF CALL2} \frac{\Gamma \triangleright e : t' \triangleleft \Gamma' \quad \Gamma' \triangleright \text{this} : C[u; \vec{F}] \quad (t \ m(t' x) \ \{s\}, 1) \in C.\text{methods}}{\Gamma \triangleright \text{this}.m(e) : t \triangleleft \Gamma'} \\
\\
\text{T-CALL} \frac{w \neq \text{this} \quad u.\text{allows}(m) = u' \quad \Gamma \triangleright e : t' \triangleleft \Gamma' \quad \Gamma' \triangleright w : C[u; \vec{F}] \quad (t \ m(t' x) \ \{s\}, \_) \in C.\text{methods} \quad \text{un}(\Gamma' \setminus \Gamma)}{\Gamma \triangleright w.m(e) : t \triangleleft \Gamma' \{w \mapsto C[u'; \vec{G}]\}}
\end{array}$$